[0023] First, in step 101, it is determined whether or not there is an array, in an array table 30, whose placement destination has not been determined. If there is such an array, the process moves to Step 102. In Step 102, referring to a dist information field 302, it is judged whether or not the array is specified by the dist directive. If the array is specified, the process moves to Step 103, and the placement destination of the array is set as a private space. In other words, "private" is registered in a placement destination field 304. In the example in FIG. 3, this step is applied to arrays a' and b'. On the other hand, in the case where the array is not specified by the dist directive, the process moves to Step 104, and the placement destination of the array is set as a shared space. In other words, "shared" is registered in the placement destination field 304. In the example in FIG. 3, this step is applied to arrays a and b of main and dsm_sub.

FIG. 1

Compile Method of the Present Invention

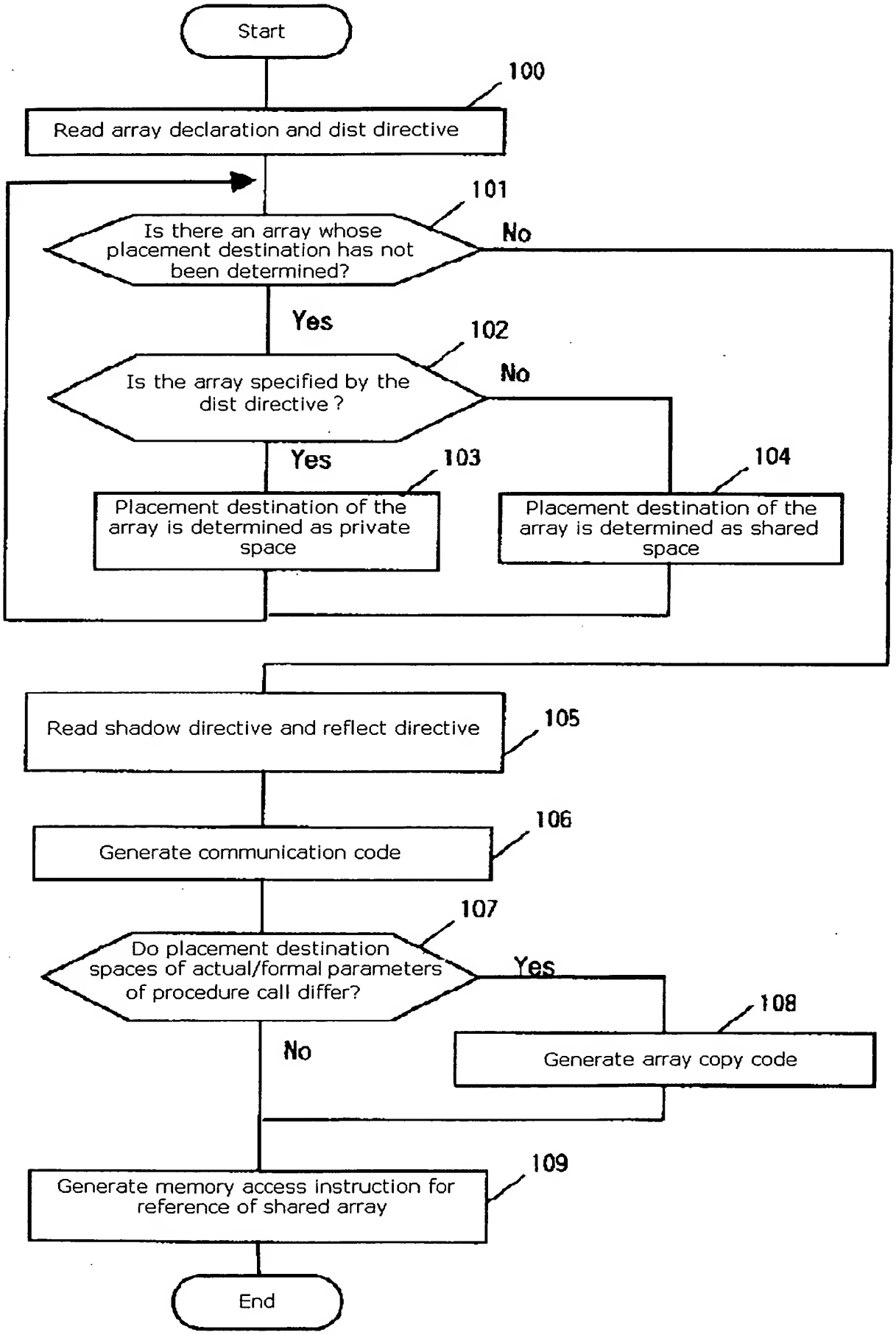


FIG. 2

Example of Input Program

20

```
 1:      program main
 2:      real a(100),b(100)
 3:      call dsm_sub(a,b)
 4:      call mp_sub(a,b)
 5:      end

11:      subroutine dsm_sub(a,b)
12:      real a(100),b(100)
13:    !$omd parallel do
14:      do i = 1,99
15:       a(i) = b(i+1) !communication via DSM
16:      enddo
17:      return
18:      end

21:      subroutine mp_sub(a',b')
22:      real a'(100),b'(100)
23:    !$omd dist(block) :: a',b'
24:    !$omd shadow(0:1) :: b'
25:    !$omd reflect(b')  !communication here
26:    !$omd parallel do
27:      do k = 1,1000
28:        do i = 1,99
29:         a'(i) = b'(i+1) ! no communication
30:        enddo
             :
31:      enddo
32:      return
33:      end
```

FIG. 3

Array Table

30

300 Array Name	301 Shape	302 dist Information	303 shadow Information	304 Placement Destination
a (main)	(100)	—	—	shared
b (main)	(100)	—	—	shared
a (dsm_sub)	(100)	—	—	shared
b (dsm_sub)	(100)	—	—	shared
a' (mp_sub)	(100)	(block)	—	private
b' (mp_sub)	(100)	(block)	(0:1)	private

PATENT ABSTRACTS OF JAPAN

(11)Publication number : 2002-073579
(43)Date of publication of application : 12.03.2002

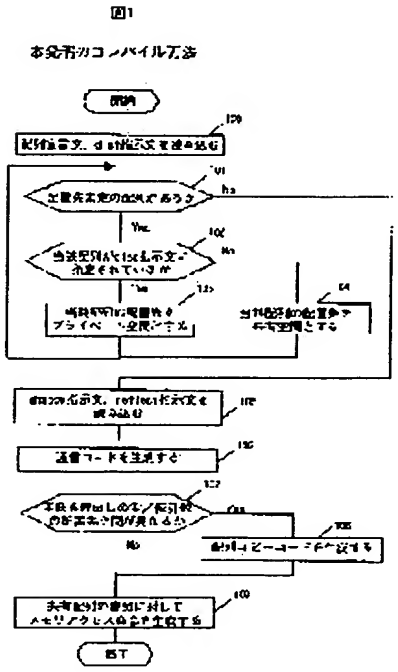
51)Int.Cl. G06F 15/16
G06F 9/45

21)Application number : 2000-264491
22)Date of filing : 29.08.2000

(71)Applicant : HITACHI LTD
(72)Inventor : OTA HIROSHI
IIZUKA TAKAYOSHI
SATO MAKOTO

54) COMPILING METHOD FOR DISTRIBUTED COMMON MEMORY

57)Abstract:
PROBLEM TO BE SOLVED: To solve such a problem that an incremental developing method for correcting a main part in a program little by little, advancing a tuning while confirming the effect and bringing out the highest performance from a successive program, has been difficult in a conventional case as the program developing method of a distributed memory-type multiprocessor.
SOLUTION: A compiler where an array is arranged in a common space or a private space for a parallel system provided with a distributed common memory mechanism, an inter-processor communication code is generated on the private array, a regular memory access instruction is generated on the common array and an array copy code between the common space and the private space is generated, is provided. A user can realize incremental program development for operating the program by arranging the array in the common space and using the distributed common memory mechanism, and arranging the array in the private space only for a kernel part so as to clearly control communication.



LEGAL STATUS

[Date of request for examination]

[Date of sending the examiner's decision of rejection]

[Kind of final disposal of application other than the examiner's decision of rejection or application converted registration]

[Date of final disposal for application]

[Patent number]

[Date of registration]

[Number of appeal against examiner's decision of rejection]

[Date of requesting appeal against examiner's decision of rejection]

[Date of extinction of right]

(19)日本国特許庁（ＪＰ）　(12)公開特許公報（Ａ）　(11)特許出願公開番号
特開2002－73579
（P2002－73579A）
(43)公開日　平成14年３月12日(2002.3.12)

(51)Int.Cl.'	識別記号	ＦＩ	テーマコード'(参考)
Ｇ０６Ｆ　15/16	６３０	Ｇ０６Ｆ　15/16	６３０Ｃ　５Ｂ０４５
9/45		9/44	３２２Ｈ　５Ｂ０８１

審査請求　未請求　請求項の数４　ＯＬ　（全 12 頁）

(21)出願番号　特願2000－264491(P2000－264491)

(22)出願日　平成12年８月29日(2000.8.29)

(71)出願人　000005108
株式会社日立製作所
東京都千代田区神田駿河台四丁目６番地
(72)発明者　太田　寛
神奈川県川崎市麻生区王禅寺1099番地　株式会社日立製作所システム開発研究所内
(72)発明者　飯塚　孝好
神奈川県川崎市麻生区王禅寺1099番地　株式会社日立製作所システム開発研究所内
(74)代理人　100075096
弁理士　作田　康夫

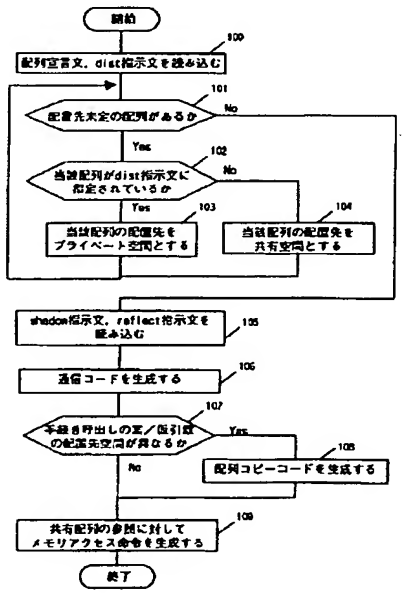
最終頁に続く

(54)【発明の名称】　分散共有メモリ向けコンパイル方法

(57)【要約】
【課題】分散メモリ型マルチプロセッサのプログラム開発手法として，従来は，逐次プログラムから出発して，プログラム中の主要部分を少しずつ修正し，効果を確かめながらチューニングを進め最終的に最高性能を引出すという，インクリメンタルな開発手法が困難であった。
【解決手段】分散共有メモリ機構を備える並列システム向けに，配列を共有空間またはプライベート空間に配置し，プライベート配列についてプロセッサ間通信コードを生成し，共有配列については通常のメモリアクセス命令を生成し，また共有空間とプライベート空間の間での配列コピーコードを生成するコンパイラを提供する。
【効果】ユーザは，まず配列を共有空間に配置して分散共有メモリ機構を用いることによってプログラムを動作させ，次にカーネル部分に対してのみ，配列をプライベート空間に配置して通信を明示的に制御するという，インクリメンタルなプログラム開発が可能になる。

図1

本発明のコンパイル方法



1

【特許請求の範囲】

【請求項１】分散共有メモリ機構を備える並列システム向けのコンパイル方法であって，配列を共有空間に配置するか各プロセッサのプライベート空間に配置するかを決定するステップと，共有空間に配置した配列の参照に対して当該配列のアドレスへのメモリアクセス命令を生成するステップと，プライベート空間に配置した配列に対してプロセッサ間通信コードを生成するステップと，共有空間とプライベート空間の間での配列コピーコードを生成するステップとを含むことを特徴とする分散共有メモリ向けコンパイル方法。

【請求項２】請求項１のコンパイル方法であって，該配列コピーコードによって実行される処理は，共有空間上のページが既にコピー先プロセッサの物理メモリに既に割り付けられているかどうかを判定するステップと，既に割り付けられている場合はコピー先の仮想プライベート空間上のページを当該物理ページにマッピングするステップとを含むことを特徴とする分散共有メモリ向けコンパイル方法。

【請求項３】分散共有メモリ機構を備える並列システム向けのコンパイル方法であって，配列の配置空間を指定する第１のユーザ指示および通信コード生成を指定する第２のユーザ指示を読み込むステップと，該第１のユーザ指示に従って配列を共有空間に配置するか各プロセッサのプライベート空間に配置するかを決定するステップと，共有空間に配置した配列の参照に対して当該配列のアドレスへのメモリアクセス命令を生成するステップと，該第２のユーザ指示に従ってプライベート空間に配置した配列に対してプロセッサ間通信コードを生成するステップと，共有空間とプライベート空間の間で配列をコピーするコードを生成するステップとを含むことを特徴する分散共有メモリ向けコンパイル方法。

【請求項４】分散共有メモリ機構を備える並列システム向けのコンパイル方法をコンピュータに実行させるための処理手順プログラムを格納するコンピュータ読み込み可能な記録媒体であって，前記方法は、

配列を共有空間に配置するか各プロセッサのプライベート空間に配置するかを決定する手順と，共有空間に配置した配列の参照に対して当該配列のアドレスへのメモリアクセス命令を生成する手順と，プライベート空間に配置した配列に対してプロセッサ間通信コードを生成する手順と，共有空間とプライベート空間の間での配列コピーコードを生成する手順とを有することを特徴とする記録媒体。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】本発明は，分散共有メモリ機構を備える並列システム向けのコンパイラに関する。

【０００２】

2

＊【従来の技術】並列計算機システムの一形態として，各プロセッサに分散したメモリを備えるものがある。このようなシステムを分散メモリ型マルチプロセッサ(DMP)と呼ぶ。例えば，プロセッサ数が100を超えるような高並列計算機や，パーソナルコンピュータ(PC)をネットワークで接続したPCクラスタなどが，この部類に属する。DMPの中には，物理的に分散されたメモリを共有メモリとしてアクセスできる分散共有メモリ(DSM: Distrbuted Shared Memory)機構を備えるものがある。すなわち，任意のプロセッサから，通常のメモリアクセス命令によって，自プロセッサに付随するメモリ(ローカルメモリ)も他のプロセッサに付随するメモリ(リモートメモリ)も区別することなくアクセスできる。DSMには，ハードウェアによって実現されるものや，OSによってソフトウェア的に実現されるものがあるが，通常のメモリアクセス命令によってアクセスできるという点は共通である。DSMの利点は，ユーザがプログラミングを行うときに，通常の共有メモリのインタフェースを用いることができる点である。すなわち，処理の並列実行を記述するだけでよく，プロセッサ間のデータ転送(通信)を気にする必要がない。このようなプログラミングインタフェースの代表的なものとしてOpenMPが知られている。OpenMPでのプログラミング例を以下に示す。

【０００３】

```
        real a(100),b(100)
!$OMP parallel do
        do i = 1,99
          a(i) = b(i+1)+...
        enddo
```

第2行目がOpenMPの指示文であり，直後のdoループの繰返しを各プロセッサに分割して，並列に実行することを指示する。ユーザはa(i)やb(i+1)でアクセスされる配列要素がどのプロセッサのメモリ上に存在するかを気にしなくて良い。

【０００４】一方，DMP向けの別のプログラミングモデルとして，メッセージ通信方式が知られている。これは，プロセッサ間でのデータの転送を，ライブラリ呼び出しの形で明示的に記述するものである。通常のメモリアクセス命令でアクセスできるのは自プロセッサのローカルメモリだけであり，リモートメモリ上のデータが必要な場合は，前もってメッセージ通信ライブラリ呼び出しによってローカルメモリに転送しておく必要がある。メッセージ通信方式の利点は，ユーザがプロセッサ間通信を明示的に制御できるため，きめ細かい性能チューニングが可能な点である。メッセージ通信方式で上記のOpenMPプログラムと同じ計算を行うためのプログラミング例を以下に示す。

【０００５】

＊

```
real a(50),b(51)
```

```
3                                                          4
  if(mypid==1) call MPI_Send(b(1),1,mypid-1)     ! データ送信
  if(mypid==0) call MPI_Recv(b(51),1,mypid+1)    ! データ受信
  if(mypid==1)  ub=49
  if(mypid==0)  ub=50
  do i = 1,ub


    a(i) = b(i+1)+...
  enddo
```

この例ではプロセッサ数は２台と仮定している。1行目の宣言文では，各プロセッサに分割された配列のサイズを宣言する必要があり，そのため配列aのサイズは50となっている。また配列bについてはさらにデータ受信用に1要素分の領域を確保する必要があり，サイズは51となっている。2，3行目では通信ライブラリ呼出しによって，後続のループでb(i+1)として参照される配列bの要素を転送している。ここで，第1引数は転送する先頭要素，第2引数は転送要素数，第3引数は転送相手のプロセッサ番号である。4，5行目では，各プロセッサが担当すべきループ範囲の上限を求めている。これらの処理の後に初めて元のループの処理が実行できる。この例に示されるように，メッセージ通信方式におけるプログラミングはきわめて煩雑になる。

【０００６】メッセージ通信方式のプログラミングの負担を軽減するための言語として，VPP Fortranが知られている。これは逐次実行用のプログラムにユーザが指示文を挿入することによって，メッセージ通信プログラムを簡潔に記述できるようにするものである。VPP Fortranで上記のメッセージ通信プログラムと同じ計算を行う例を以下に示す。

【０００７】

```
      real a(100),b(100)
!xocl processor p(2)
!xocl index partition q=(proc=p,index=1:100)
!xocl local a (/q), b(/q(overlap=(0:1))
!xocl overlapfix(b)          ! データ転送
!xocl spread do /q
    do i = 1,99
      a(i) = b(i+1)+...
    enddo
```

1行目の宣言文では，OpenMPプログラムと同様に，配列a,bのサイズは100のままでよい。2行目以降が指示文である。2行目はプロセッサが2台であることを宣言し，3行目，4行目は，配列を2台のプロセッサ上に分散配置することを指示する。さらに配列bについては，4行目のoverlap(0:1)という指示により，各プロセッサでデータ受信用の領域を1要素分確保することを指示している。5行目のoverlapfix指示文は，この受信用領域に対して隣接プロセッサ間で配列要素を転送することを指示している。6行目のspread do指示文は，後続のループをプロセッサ2台で分割して実行することを指示している。このとき，b(i+1)として参照される要素は，上記のoverlapfix指示文によって既に転送完了している。この例からも分かるように，VPP Fortranを使うと，通常の逐次プログラムに指示文を追加することによって通信が記述できるので，メッセージ通信プログラムの作成が容易になるという利点がある。

【０００８】VPP Fortranについては，岩下英俊，進藤達也，岡田信，"VPP Fortran: 分散メモリ型並列計算機向けプログラミング言語"，情報処理学会論文誌，Vol.36，No．7，pp．1542-1550 (1995) に述べられている。

【０００９】

【発明が解決しようとする課題】従来の方法では，それぞれ以下のような問題があった。

【００１０】DSMを用いたOpenMPプログラムでは，プロセッサ間通信はハードウェアやOSによって暗黙に行われるので，ユーザは通信を明示的に制御できなかった。また，データの各プロセッサの分散配置は，固定サイズのページを単位として行われるので，ユーザの要求するデータ分散が正確に実現できない場合があった。これらの制限のため，ユーザがプログラムをチューニングして最高性能を引出すのには限界があった。

【００１１】一方メッセージ通信方式では，逐次プログラムから出発して並列プログラムを作成する場合に，効果を確かめながら少しずつチューニングするという「インクリメンタル」な開発手法を取ることが困難であった。例えば，ある配列を分散した場合，その配列を参照するすべてのサブルーチンにおいて，配列サイズの変更とそれに伴う通信の挿入やループ繰返し範囲の変更を行わなければならなかった。これは事実上，プログラム全体の修正を意味する。VPP Fortranを用いた場合でも同様に，ある配列を分散した場合，関連するサブルーチンをすべて修正する必要があった。

【００１２】なおVPP Fortranでは配列に対して「グローバル属性」を指定することができる。この属性を持つ配列は，たとえ分散されていたとしても，コンパイラが自動的に実行時ライブラリを挿入してリモートメモリ参照かどうかを実行時に判定するようなコードを生成する。これを利用すれば，ある配列を分散したときに，ユーザがプログラム全体について通信を挿入する負担は軽減される。しかし，この場合，配列要素参照のたびにラ

イブラリ呼出しのオーバヘッドが発生してしまい，非常に遅くなってしまうという問題があった。

【００１３】以上をまとめると，従来は，逐次プログラムから出発してインクリメンタルにDMPプログラムを作成することが困難であった。すなわち，逐次プログラムをまずDMP上でそのまま走らせ，その後，プログラム中の主要部分を少しずつDMP向けに修正していき，効果を確かめながら並列化およびチューニングを進め，最終的に最高性能を引出すという開発手法を行うことができなかった。これは，大規模ステップ数のプログラムをDMP向けに移植する際に大きな障壁となっていた。

【００１４】本発明の目的は，DMP向けのインクリメンタルなプログラミングモデルを提供することにある。

【００１５】

【課題を解決するための手段】上記課題の解決のため，本発明では，DSM機構を備える並列システム向けのコンパイラにおいて，配列を共有空間または各プロセッサのプライベート空間のいずれかに配置し，プライベート空間に配置した配列については明示的なプロセッサ間通信コードを生成し，共有空間に配置した配列については通常のメモリアクセス命令を生成する。さらに，一つの配列をプログラム中のある個所では共有空間に置き，別の個所ではプライベート空間に置いて，共有空間とプライベート空間の間での配列コピーコードを生成する。

【００１６】このコンパイラを利用することにより，ユーザは，まず配列を共有空間に配置してDSM機構を用いることによってプログラムを動作させ，次にプログラム中のカーネル部分に対してのみ，配列をプライベート空間にコピーすることによって，通信を明示的に制御しかつ正確なデータ分散を実現するという，インクリメンタルなプログラム開発が可能になる。

【００１７】本発明ではさらに，共有空間からプライベート空間への配列コピーにおいて，仮想共有空間上のページが既にコピー先プロセッサの物理メモリに割り付けられているかどうかを判定し，既に割り付けられている場合はページの内容をコピーする代わりに，コピー先の仮想プライベート空間上のページを当該物理ページにマッピングする。これにより，データ移動量を最小限に抑え，またメモリ消費量を削減できる。

【００１８】

【発明の実施の形態】図1は，本発明のコンパイル方法の一実施形態におけるコンパイラの処理を示すフローチャートである。図2は本コンパイラへの入力プログラムの例である。図1の各ステップを説明する前に，図2のプログラムの概要を説明する。

【００１９】図2のプログラムはmain(1行目～5行目)，dsm_sub(11行目～18行目)，mp_sub(21行目～3行目)の三つのルーチンから構成される。mainの2行目は配列の宣言文であり，各々100個の要素からなる二つの配列aおよびbを宣言している。これらの配列は，mainの3行目および4行目で，dsm_subやmp_subに引数として渡され，各ルーチンの中でこれらの配列を用いた計算が行われる。dsm_subは計算量の比較的少ないルーチンであり，ユーザにとっては最高性能を追求する必要は少ない。一方mp_subは2行目に繰返し数が1000のループを含み，計算量が大きい。したがって，ユーザはmp_subに関しては最高性能を得るために十分なチューニングを行いたい。!$omdで始まる行は，チューニング用の指示文であり，本発明のコンパイラによって認識される。これらの指示文の意味は後で説明する。

【００２０】図1に戻り，図2のプログラムへの適用を例として，各ステップを説明する。プロセッサ数は2台とする。

【００２１】ステップ100では，入力プログラム中の配列宣言文およびdist指示文を読み込み，情報を図3に示す配列表30に登録する。配列宣言文は，mainの2行目，dsm_subの12行目，mp_subの22行目である。これらの宣言文を読み込んで，配列表30の配列名フィールド300，形状フィールド301に情報を登録する。dist指示文は，mp_subの23行目である。この指示文は，配列を均等なブロックに分割して，各プロセッサに割り当てること指示するものである。すなわち，2プロセッサの場合は，配列a'およびb'を50要素ずつに分割して，各プロセッサに割り当てる。コンパイラはdist指示文を読み込んで，配列表30のdist情報フィールド302に情報を登録する。なお，mainやdsm_sub内の配列aおよびbにはdist指示文が無いので，それらに対するdist情報フィールド302には何も登録しない。

【００２２】ステップ101から104では，配列表30を参照して各配列を共有空間に配置するかプライベート空間に配置するかを決定する。ここで共有空間およびプライベート空間とは，共に仮想メモリ空間上の領域であり，共有空間はDSM機構を利用して各プロセッサからアクセスできる領域，プライベート空間は各プロセッサに固有の領域であり他プロセッサからはアクセスできないものである。

【００２３】まずステップ101では配列表30の中に配置先が未定の配列があるかどうかを判定し，あればステップ102に進む。ステップ102では，dist情報フィールド302を参照して，当該配列がdist指示文で指定されているかどうかを判定する。もし指定されていればステップ103に進んで，当該配列の配置先をプライベート空間と決定する。すなわち，配置先フィールド304に「private」を登録する。図3の例では，配列a'およびb'について，本ステップが適用されている。一方dist指示文で指定されていない場合にはステップ104に進んで，当該配列の配置先を共有空間と決定する。すなわち，配置先フィールド304に「shared」を登録する。図3の例では，mainおよびdsm_sub内の配列aおよびbについて，本ステップが適用されている。

【００２４】以上の手順によってすべての配列に対して配置先が決定したら，ステップ105に進む。

【００２５】ステップ105では，shadow指示文およびreflect指示文を読み込む。図2では，shadow指示文はmp_subの24行目，reflect指示文は同じくmp_subの25行目である。このうちshadow指示文は，各プロセッサに分散された配列の端に受信用のバッファを指定された要素数だけ設けることを指示する。例えば図2のshadow指示文は，分散された配列b'の両側に，それぞれ0要素分および1要素分のバッファを設けることを意味する。コンパイラはshadow指示文を読み込んで，配列表30のshadow情報フィールド303に情報を登録する。

【００２６】一方，reflect指示文はshadow指示文で指定された受信バッファに対して，実際にデータ転送を実行することを指示する。図2のプログラムでは，配列b'は1要素分のshadow領域を持つので，分散された配列の端の１要素を隣接プロセッサに転送して，該隣接プロセッサのshadow領域に格納する。

【００２７】ステップ106では，前ステップで読み込んだreflect指示文に基づいて，実際に通信コードを生成する。この処理は実際にはコンパイラ中間語に対して行われるが，ここでは図4に示すようなソースプログラムの形式で説明する。

【００２８】図4は，図2の入力プログラムに対する出力プログラムを，ソースプログラム形式で表現したものである。図1のステップ106で生成する通信コードは，83行目および84行目である。ここでOMD_Sendは，第1引数および第2引数で指定された自プロセッサ上のデータを，第3引数で指定されたプロセッサに送信するライブラリルーチンとする。第1引数は転送する先頭要素，第2引数は転送要素数を表す。またOMD_Recvは，第3引数で指定されたプロセッサからデータを受信して，第1引数および第2引数で指定された自プロセッサ上の領域に格納するライブラリルーチンとする。mypidは自プロセッサ番号を表す。したがって，83行目は，プロセッサ1(以下P1)がプロセッサ0(以下P0)に対して自身のプライベート空間内の配列要素b'(1)を送信する処理を表し，84行目は，P0がP1から受信したデータを自身のプライベート空間内のb'(51)に格納する処理を表す。

【００２９】図1に戻って，ステップ107では，入力プログラム中の手続き呼出しを検出して，各実引数について，対応する仮引数の配置先空間が実引数と異なるかどうかを判定する。もし異なる場合，すなわち，一方が共有空間でもう一方がプライベート空間の場合には，ステップ108に進む。例えば，図2の入力プログラムでは，mainの3行目でdsm_subを呼出しているが，実引数も仮引数も共に共有空間に配置されているから条件に該当しない。一方，mainの4行目のmp_subの呼出しでは，実引数aおよびbは共有空間に配置されているが，対応する仮引数a'およびb'は共にプライベート空間に配置されている。したがって，この手続き呼出しについては，ステップ108に進むことになる。

【００３０】ステップ108では，当該引数に対して，共有空間とプライベート空間の間でコピーを行うコード(以下，配列コピーコード)を生成する。例えばmp_subの呼出しに対しては，図4の出力プログラムの55行目から59行目に示されるように，mp_sub呼出しの前後に，OMD_array_copyというライブラリ呼出しを生成する。ここでOMD_array_copyは第1引数で指定された配列を第2引数で指定された配列にコピーするライブラリルーチンである。実際にはコピーする範囲などを表す引数が続くが，ここでは省略してある。55行目および56行目によって，共有空間の配列aおよびbからプライベート空間の配列a'およびb'へのコピーが実行される。またmp_sub呼出しから戻った後に58行目および59行目によって，逆にプライベート空間から共有空間への配列コピーが実行される。

【００３１】ステップ109では，共有空間に配置された配列の要素の参照に対して，当該配列要素のアドレスへのメモリアクセス命令を生成する。図4に示す出力プログラムは実際には機械語コードの形で生成されるが，そこでは配列の参照はメモリアクセス命令に置換されている。この命令を生成するのが本ステップである。例えばdsm_sub内での配列要素b(i+1)の参照に対して，図5に示すような命令列を生成する。図5において，r0，r1，などはレジスタを表す。レジスタr0にはbの先頭アドレス，r1にはiの値が格納されているものとする。これらの値を元に3行目のadd命令から6行目のadd命令までででb(i+1)のアドレスを計算し，レジスタr2に格納している。最後のld命令がメモリアクセス命令であり，r2で指定されるアドレスの内容，すなわちb(i+1)の値をr3にロードしている。

【００３２】以上で，図1のコンパイル方法の説明を終わり，これ以降は，ステップ108で生成した配列コピーコードの処理内容を，より詳細に説明する。

【００３３】図4の出力プログラムは，各々のプロセッサによって実行されるものである。52行目で宣言されている配列aおよびbは共有空間に配置されているので，両方のプロセッサから同じものが見えるが，53行目で宣言されている配列a'およびb'はプライベート空間に配置されているので，それぞれのプロセッサで異なるものが見える。この様子を図6を用いて説明する。

【００３４】図6は，図4のプログラムを実行するときの，配列のメモリ配置を示す。プログラム内で宣言されている配列は，図の中心に示されるように仮想空間上に配置される。仮想空間は共有空間とプライベート空間に分かれており，プライベート空間はさらに各プロセッサに固有の空間に分かれている。また，仮想空間は，図の両側に示される各プロセッサの物理空間にマッピングされている。このマッピングはページと呼ばれる固定サイズを最小単位として行われる。ここでは，ページのサイ

ズは配列要素20個分であると仮定する。

【００３５】配列aおよびbは全体(100個要素)が共有空間に配置される。これは，ページを単位として物理空間にマッピングされている。どのページがどのプロセッサにマッピングされるかは任意であるが，いずれにしても，100要素は5ページに相当しプロセッサ数で割り切れないので，マッピングされる要素数はプロセッサによって偏りがあることになる。ここでは，前の3ページはP0に，後の2ページはP1にマッピングされているものとする。

【００３６】また，配列a'およびb'は各プロセッサのプライベート空間に，それぞれ50要素または51要素ずつ配置される。配列b'の最後の(51番目)の要素は，shadow領域である。これらは，各プロセッサの物理空間にマッピングされなければならない。マッピングはやはりページ単位で行われるが，各プロセッサのプライベート空間は仮想空間上で元々独立しているので，共有空間のように偏りが生じることなく，そのままの大きさで物理空間にマッピングされる。

【００３７】前述のライブラリルーチンOMD_array_copyの中では，自プロセッサのプライベート空間と対応する共有空間との間で配列をコピーする。例えば，図4の55行目の

```
call OMD_array_copy(a,a',...)
```

では，P0は共有空間のa(1:50)を自身のプライベート空間のa'(1:50)にコピーし，P1は共有空間のa(51:100)を自身のプライベート空間のa'(1:50)にコピーする。また，図4の56行目の

```
call OMD_array_copy(b,b',...)
```

では，配列bおよびb'について同様の処理を行う。58行目および59行目では逆方向のコピーを実行する。

【００３８】以上が基本的な配列コピー方法であるが，これをさらに改良した別の方法もある。この別方法を以下に説明する。

【００３９】図7に，別方法における配列のメモリ配置を示す。本方法では，仮想プライベート空間を予め物理空間へマッピングしておくのではなく，配列コピーコードの中でマッピングを行う。このとき，仮想共有空間上のコピーすべきページが既に自プロセッサの物理空間にマッピングされているならば，内容をコピーする代わりにコピー先仮想プライベート空間のページを当該物理空間のページにマッピングする。例えば，図7において，共有空間の配列bの最初の2ページは，コピー先がP0のプライベート空間であり，かつ，既にP0の物理空間にマッピングされているから，P0のプライベート空間の配列b'の最初の2ページを当該物理ページにマッピングする。b'の残りの部分，すなわち最後の10要素および1要素分のshadow領域については，新たな物理ページを確保してそこにマッピングする。P1のプライベート空間の配列b'については，後の2ページが既にP1の物理空間にマッピングされているので，これらについては当該物理ページにマッピングする。残りの部分，すなわち最初の10要素および最後のshadow領域については，それぞれ新たな物理ページを確保してそこにマッピングする。最初の10要素とshadow領域とはメモリ上連続でないので，別々のページにマッピングしなければならない。

【００４０】図8は，図7のメモリ配置を用いる場合の配列コピー方法を表すフローチャートである。これはライブラリルーチンOMD_array_copyの中で共有空間からプライベート空間へ配列をコピーする場合に実行される処理である。

【００４１】図8のステップ120では，共有空間上のコピーすべき配列を構成する各ページにつき，コピー処理が済んでいるかどうかを判定し，未処理であれば当該ページに対してステップ121以降を実行する。

【００４２】ステップ121では，当該ページに含まれる全要素についてコピー先プロセッサが同一かどうかを判定する。これは必ずしも個々の要素について一つずつ判定する必要はなく，より簡単な判定，例えば，プロセッサへの分割境界のインデックス(図7では50)がページ内に含まれるかどうかの判定を行えば十分である。もしコピー先プロセッサが同一だと判明したらステップ122に進み，そうでない場合はステップ124に進む。

【００４３】ステップ122では，当該ページが既にコピー先プロセッサの物理空間にマッピングされているかどうかを判定する。もしそうならばステップ123に進み，さもなくばステップ124に進む。

【００４４】ステップ123では，コピー先のプライベート空間の当該仮想ページを，ステップ122で求められた物理ページにマッピングする。

【００４５】一方，ステップ124に進んだ場合は，コピー先プロセッサの物理空間に新たなページを確保する。そしてステップ125で，コピー先プライベート空間の仮想ページを当該物理ページにマッピングする。さらにステップ126で，共有空間のページに含まれる配列要素を確保した当該物理ページに実際にコピーする。

【００４６】以上で配列コピー処理の別方法の説明を終わる。

【００４７】最後に，図1のコンパイル方法を実行するコンパイラの構成を示す。図9はそのようなコンパイラの構成の一例である。コンパイラ4には，構文解析部40，配列配置先空間決定部41，中間語変換部42，および出力プログラム生成部43が含まれる。

【００４８】構文解析部40は，入力プログラム20を読み込んで，中間語50を生成する。入力プログラムの内容は例えば図2に示したようなものである。中間語50はコンパイラ内部でのプログラムの表現形式であり，その形式は通常のコンパイラの場合と特に変わらないので，ここでは特に詳細は述べない。構文解析部の中に含まれる指示文解析部400は，指示文を読み込む処理を実行する。

すなわち，図1のステップ100，およびステップ105の処理を実行し，配列表30に情報を登録する。

【００４９】配列配置空間決定部41は，配列表30の中の指示文情報を参照して，配列を共有空間に配置するかプライベート空間に配置するかを決定する。これは図1のステップ102からステップ104の処理である。

【００５０】中間語変換部42には，通信コード生成部420および配列コピーコード生成部421およびメモリアクセス命令生成部422が含まれる。通信コード生成部420は，中間語内のreflect指示文にしたがって，通信コードの中間語を生成する。これは図1のステップ106の処理である。配列コピーコード生成部421は，手続き呼出しの実引数と仮引数の配置先空間に応じて，配列コピーコードの中間語を生成する。これは図1のステップ108の処理である。メモリアクセス命令生成部422は，共有空間に配置された配列の参照に対してメモリアクセス生成する。これは図1のステップ109の処理である。

【００５１】出力プログラム生成部43は，変換された中間語50を読み込んで，出力プログラム21を生成する。出力プログラム21の内容は例えば図4に示したようなものである。本実施形態の説明では図4のように高級言語ソースプログラムの形式を用いて説明したが，代わりに機械語形式を用いることもできる。出力プログラム生成部43の内容は通常の特に変わらないので，これ以上の詳細は述べない。

【００５２】以上で，本発明の実施形態の説明を終わる。

【００５３】

【発明の効果】本発明によれば，ユーザは，まず配列を共有空間に配置してDSM機構を用いることによってプログラムを動作させ，次に，プログラム中のカーネル部分＊に対してのみ，最高性能を引出すためのチューニングを行うという，インクリメンタルなプログラム開発が可能になる。また，本発明によればさらに，上記チューニングによって発生する配列コピーにおいて，データ移動量やメモリ消費量を最小限に抑え，プログラムの実行性能を向上させることができる。

【図面の簡単な説明】

【図１】本発明のコンパイル方法の一例を示すフローチャートである。

【図２】本発明のコンパイラへの入力プログラムの例である。

【図３】コンパイラ内部で利用される配列表である。

【図４】図2の入力プログラムに本発明を適用したことによって得られる出力プログラムである。

【図５】図4の出力プログラムにおける配列要素の参照に対する機械語の命令列である。

【図６】本発明における配列のメモリ配置を示す図である。

【図７】本発明における配列のメモリ配置の別方法を示す図である。

【図８】図７のメモリ配置を用いた場合の，共有空間からプライベート空間への配列コピー処理を示すフローチャートである。

【図９】本発明のコンパイル方法を実施するコンパイラの構成例である。

【符号の説明】

4…コンパイラ，20…入力プログラム，21…出力プログラム，30…配列表，400…指示文解析部，41…配列配置空間決定部，420…通信コード生成部，421…配列コピーコード生成部，422…メモリアクセス命令生成部。

【図３】

図3

配列表

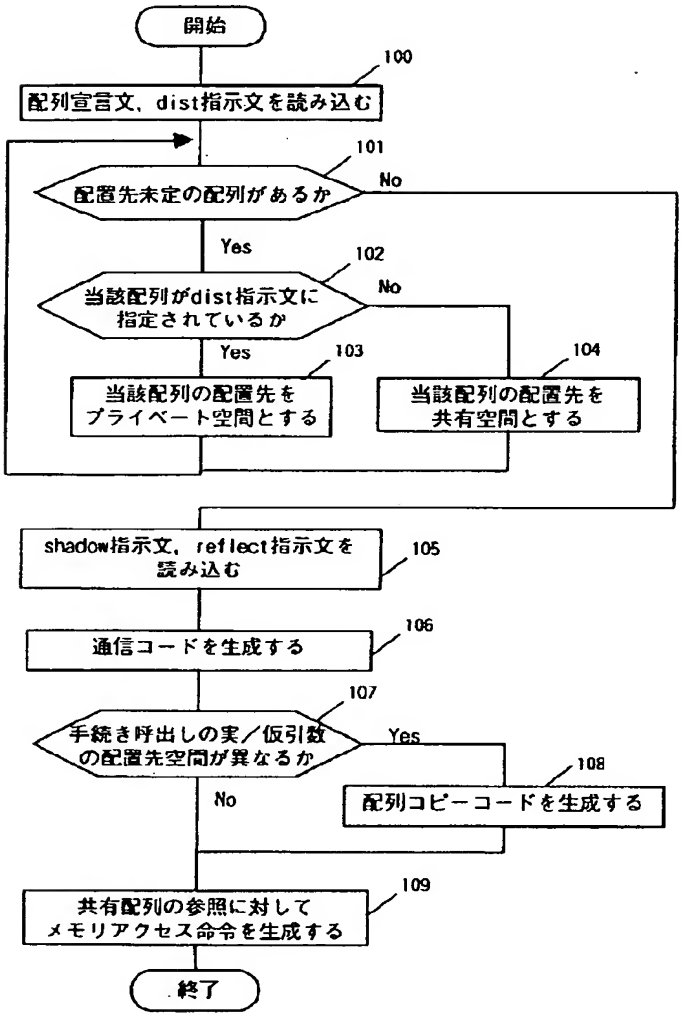
30

300 配列名	301 形状	302 dist情報	303 shadow情報	304 配置先
a (main)	(100)	—	—	shared
b (main)	(100)	—	—	shared
a (dsm_sub)	(100)	—	—	shared
b (dsm_sub)	(100)	—	—	shared
a' (mp_sub)	(100)	(block)	—	private
b' (mp_sub)	(100)	(block)	(0:1)	private

【図１】

図1

本発明のコンパイル方法



【図２】

図2

入力プログラム例

20

```
 1:     program main
 2:     real a(100),b(100)
 3:     call dsm_sub(a,b)
 4:     call mp_sub(a,b)
 5:     end

11:     subroutine dsm_sub(a,b)
12:     real a(100),b(100)
13:   !$omd parallel do
14:     do i = 1,99
15:       a(i) = b(i+1) ! DSMによる通信
16:     enddo
17:     return
18:     end

21:     subroutine mp_sub(a',b')
22:     real a'(100),b'(100)
23:   !$omd dist(block) :: a',b'
24:   !$omd shadow(0:1) :: b'
25:   !$omd reflect(b')  ! ここで通信
26:   !$omd parallel do
27:     do k = 1,1000
28:       do i = 1,99
29:         a'(i) = b'(i+1)  ! 通信無し
30:       enddo
            :
31:     enddo
32:     return
33:     end
```

【図４】

図4

出力プログラム例

21

```
51:     program main
52:     real a(100),b(100)      ! 共有空間
53:     real a'(50),b'(51)      ! プライベート空間
54:     call dsm_sub(a,b)
55:     call OMD_array_copy(a,a',...)   ! 配列コピー
56:     call OMD_array_copy(b,b',...)   ! 配列コピー
57:     call mp_sub(a',b')
58:     call OMD_array_copy(a',a,...)   ! 配列コピー
59:     call OMD_array_copy(b',b,...)   ! 配列コピー
60:     end

71:     subroutine dsm_sub(a,b)
72:     real a(100),b(100)
73:     do i = ...      ! 自プロセッサの担当範囲
74:      a(i) = b(i+1) ! DSMによる通信
75:     enddo
76:     return
77:     end

81:     subroutine mp_sub(a',b')
82:     real a'(50),b'(51)
83:     if(mypid==1) call OMD_Send(b'(1),1,0)   ! 送信
84:     if(mypid==0) call OMD_Recv(b'(51),1,1)  ! 受信
85:     do k=1,1000
86:       do i = ...          ! 自プロセッサの担当範囲
87:        a'(i) = b'(i+1)   ! 通信無し
88:       enddo
            :
89:     enddo
90:     return
91:     end
```

【図５】

図5

共有配列参照に対する命令列

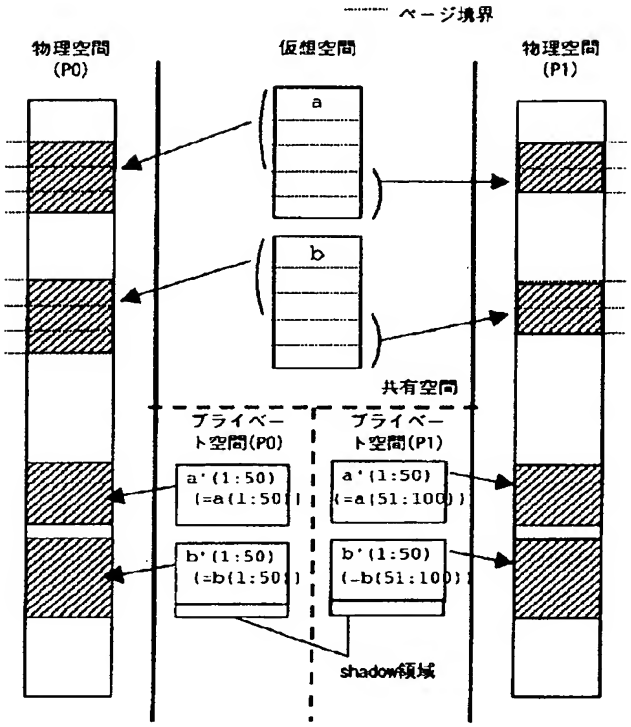
22

```
dsm_sub:
        :                   ; r0: bの先頭アドレス  r1: iの値
      add   r1,1,r2         ; r2 ← i+1
      sub   r2,1,r2         ; r2 ← i+1-1
      mul   r2,4,r2         ; r2 ← (i+1-1)*4
      add   r0,r2,r2        ; r2 ← r0+(i+1-1)*4
      ld    [r2] r3         ; r2で指定されるアドレスをr3にロード
        :
```

【図６】

図6

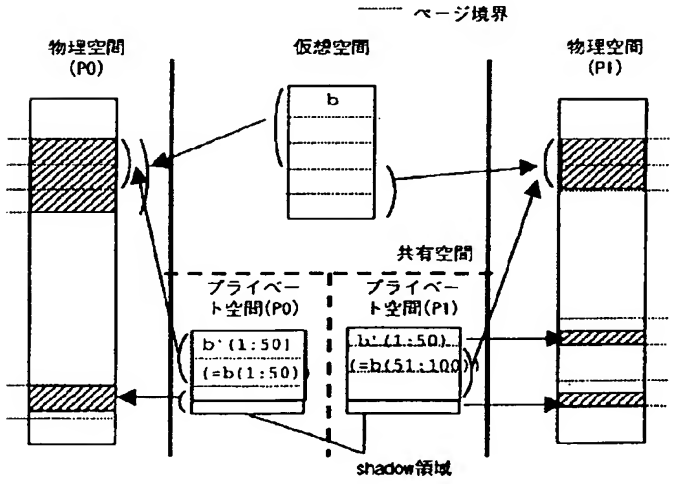
配列のメモリ配置



【図７】

図7

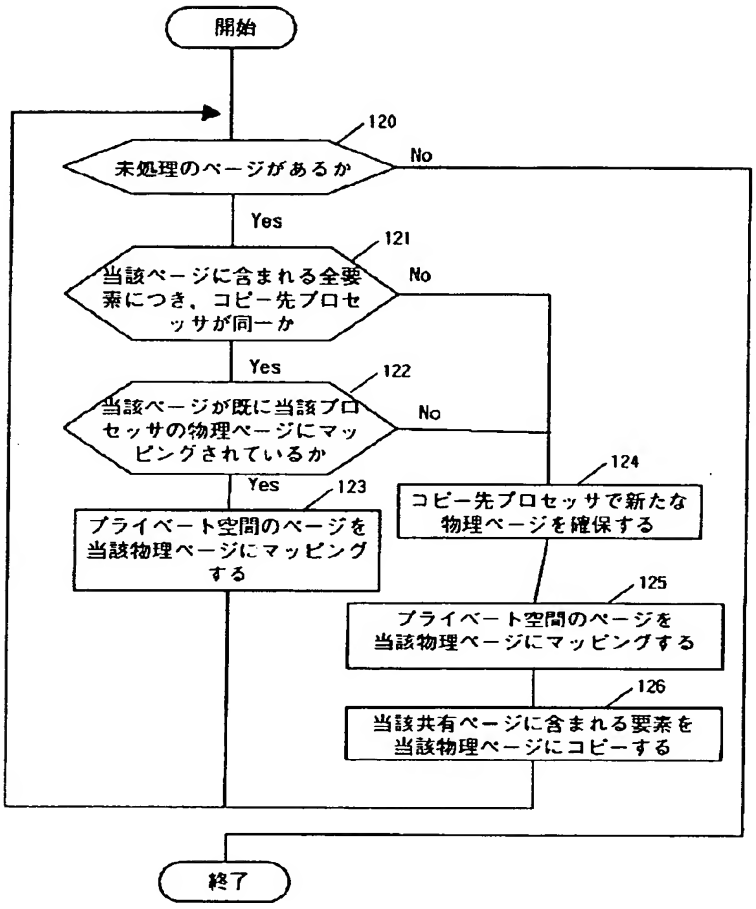
配列のメモリ配置の別方法



【図８】

図8

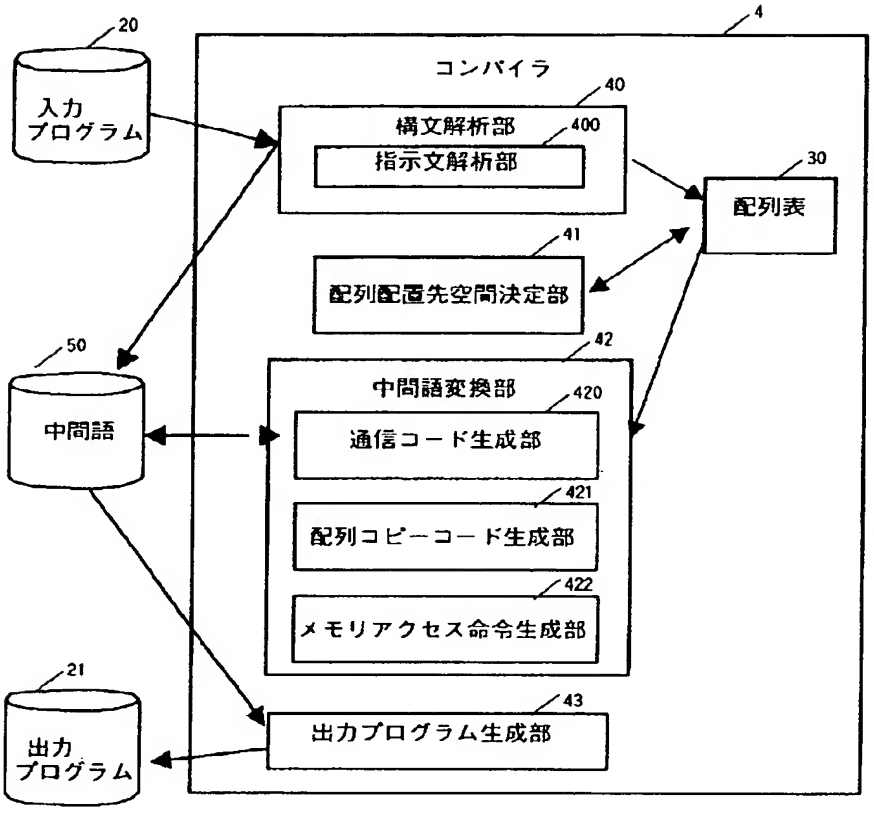
配列コピーの別方法



【図９】

図9

コンパイラの構成



フロントページの続き

(72)発明者　佐藤　真琴
神奈川県川崎市麻生区王禅寺1099番地　株式会社日立製作所システム開発研究所内

Ｆターム(参考)　5B045 DD04 GG11
5B081 BB08 CC28 CC29 CC32